



Des correctifs de sécurité à la mise à jour

Nicolas Lorient, Marc Ségura Devillechaise, Jean-Marc Menaud

► To cite this version:

Nicolas Lorient, Marc Ségura Devillechaise, Jean-Marc Menaud. Des correctifs de sécurité à la mise à jour. 2004, pp.65-76. hal-00003283

HAL Id: hal-00003283

<https://hal.science/hal-00003283>

Submitted on 24 Nov 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Des correctifs de sécurité à la mise à jour

Audit, déploiement distribué et injection à chaud

Nicolas Lorient - Marc Ségura-Devillechaise - Jean-Marc Menaud

École des Mines de Nantes, projet Obasco EMN/INRIA,
{nlorient,msegura,jmenaud}@emn.fr <http://www.emn.fr/x-info/arachne>

RÉSUMÉ. La complexité toujours croissante des logiciels implique un accroissement des bogues. Ces derniers restent le vecteur principal d'attaque des pirates informatiques et imposent aux éditeurs logiciels d'émettre régulièrement des mises à jour pour leurs corrections. Cependant, peu d'utilisateurs sont prêts à arrêter leurs applications et à investir le temps nécessaire pour mettre à jour leur poste de travail. Dans les faits, les correctifs sont rarement déployés, même quand ceux-ci sont critiques.

Nous proposons un atelier de déploiement permettant à un administrateur système d'appliquer à chaud, à distance et sans l'intervention des utilisateurs, des rustines de sécurité. Pour éviter l'arrêt des logiciels concernés, notre approche est basée sur un tisseur dynamique d'aspects, Arachne. Notre second outil, Minerve, s'intègre dans le processus de mise à jour standard : il prend en entrée une rustine source produite par `diff` et génère, quand cela est possible, une rustine dynamique déployable à la volée. Il permet de consulter les rustines exprimées dans un langage dédié et facilite ainsi la tâche d'audit des mises à jour déployées.

ABSTRACT. The ever growing software complexity suggests that they will never be bugfree and therefore secure. Software companies regularly publish updates. But maybe because of lack of time or care or maybe because stopping application is annoying, such updates are rarely if ever deployed on users' machines.

We propose an integrated tool allowing system administrators to deploy critical security updates on the fly on applications running remotely without end-user intervention. Our approach is based on an aspect weaving system, Arachne, that dynamically rewrites binary code. Hence updated applications are still running while they are updated. Our second tool Minerve integrates Arachne within the standart updating process: Minerve takes a patch produced by `diff` and eventually builds a dynamic patch that can later be woven to update the application on the fly. In addition, Minerve allows to consult patches translated in a dedicated language and hence eases auditing tasks.

MOTS-CLÉS : mise à jour de programme à chaud, programmation par aspect, déploiement distribué, sécurité

KEYWORDS: hot software update, aspect-oriented programming, distributed deployment, security

1. Introduction

Pour l'année 2003, 80% des attaques informatiques subies par les entreprises sont liées à un manquement dans le déploiement des correctifs de sécurité[DEV 04]. En effet, la plupart des attaques informatiques exploitent des failles de sécurité répertoriées. Dans le cas de la dernière attaque médiatisée, le virus Sasser, une rustine corrigeant la vulnérabilité exploitée était disponible quinze jours avant la propagation du virus. De ce fait, une grande partie des attaques informatiques subies pourraient être évitées en menant de front à la fois, une veille attentive sur les failles de sécurité détectées par les organismes spécialisés, et par une application rapide des correctifs sur l'ensemble des machines concernées. Cette double mission est généralement octroyée à un administrateur système promu au rang de RSSI (Responsable de la Sécurité du Système Informatique). En l'absence d'outils adéquats, le RSSI est toutefois dénué de moyens pour lutter efficacement contre les d'attaques potentielles. En effet, la simple mission de veille lui demandera un peu plus de 13 semaines de travail à raison de cinq minutes pour chacune des 5500 alertes annuelles recensées[CER 03]. En supposant qu'il ait la charge d'administrer 100 machines, qu'elles ne soient concernées que par 1% des alertes et que l'application d'un correctif demande une heure de travail, le temps nécessaire à leur déploiement s'élèvera au total à plus de 157 semaines de travail par an.

Nous proposons dans cet article une approche semi-automatique pour le déploiement des correctifs de sécurité. Le but de cette approche est de réduire significativement le temps nécessaire au déploiement des correctifs. Le principal problème n'est pas tant l'application du correctif en lui même mais d'obtenir de l'utilisateur l'accès aux machines nécessaires et l'arrêt des applications à mettre à jour (prise de contact, sauvegarde de la session et des documents etc.). Notre approche est basée sur deux outils, Minerve et Arachne. Le premier réduit le temps de certification des correctifs par le RSSI. Le second permet le déploiement des correctifs sans avoir à arrêter l'application à mettre à jour. La mise à jour à chaud évite au RSSI d'obtenir l'accord de l'utilisateur pour appliquer un correctif et réduit ainsi significativement leur temps de déploiement. La section 2 présente une vue d'ensemble de notre atelier de déploiement et son intégration dans le processus traditionnel de mise à jour des logiciels libres. Les sections 3 et 4 présentent respectivement nos outils : Minerve et Arachne. La section 5 décrit une évaluation de l'atelier appliqué aux derniers trous de sécurité. Avant de conclure, la section 6 résume quelques principaux travaux connexes.

2. Atelier de déploiement

Traditionnellement dans les logiciels libres, lorsqu'une faille de sécurité est découverte, un correctif est proposé sous la forme d'une rustine source (patch) produite par l'outil `diff`. Cet outil permet de lister les différences textuelles entre deux fichiers sources.

Lorsque le RSSI prend connaissance d'une faille de sécurité qui affecte son parc

informatique, il vérifie en examinant les sources du logiciel que le correctif ne remet pas en cause son intégrité. Il doit alors recompiler, redéployer le logiciel corrigé sur l'ensemble du parc puis ré-initialiser le logiciel et/ou la machine.

Ce processus de mise à jour présente plusieurs lacunes. En premier lieu, l'audit des rustines fournies par les éditeurs logiciels est difficile à réaliser car les fichiers de différences textuelles entre les versions d'un logiciel ne fournissent pas d'informations pertinentes et claires des modifications apportées par la rustine. En second lieu, les failles de sécurité sont toujours présentes tant que le logiciel incriminé n'a pas été stoppé. Or pour certaines applications comme les serveurs de vente en ligne, cette contrainte n'est pas acceptable. De plus, un RSSI ne peut prendre la responsabilité d'arrêter l'application d'un utilisateur sans savoir si ce dernier a préalablement sauvegardé son travail. En dernier lieu, le déploiement du logiciel corrigé doit se faire machine après machine ce qui, sans outil adapté, est une perte de temps importante et peut être compliqué dans un environnement mobile.

Notre atelier logiciel est destiné à faciliter la lecture des correctifs, à automatiser leurs déploiements et enfin à les appliquer dynamiquement sans arrêt des logiciels. Dans ce dessein, notre atelier est basé sur deux outils : Minerve et Arachne.

Le premier est chargé à la fois de valider le correctif dans le cadre d'une application dynamique et de transformer les rustines sources très peu lisibles en fichiers de description des modifications présentant au RSSI une vue synthétique des modifications qui seront apportées. Minerve permet de présenter au RSSI le corps des fonctions avant et après modification en utilisant une symbolique pour désigner les points de changement. Après validation du RSSI en fonction des informations fournies par Minerve sur l'applicabilité dynamique du correctif, le fichier est compilé en binaire sous forme d'une librairie, puis déployé sur l'ensemble des machines concernées par la mise à jour.

Le second, préinstallé sur l'ensemble du parc, est alors chargé d'appliquer le correctif. Sa principale fonction consiste à s'insérer dynamiquement et sans l'arrêter dans le logiciel incriminé, puis de récrire le code binaire du logiciel en vue de remplacer les fonctions impliquées dans la faille de sécurité par leurs nouvelles versions contenues dans la DLL précédemment diffusée.

Finalement, le correctif n'est appliqué que pour la durée d'exécution du logiciel. En effet, en parallèle à cette opération, une mise à jour classique par application du patch sur les sources est effectuée. Les prochaines exécutions du logiciel seront donc liées à la nouvelle version [HP 04]. Dans notre atelier, le travail de l'administrateur se limite à effectuer une veille sur les failles et correctifs diffusés, valider les correctifs concernés par les machines/logiciels qu'il administre et finalement indiquer à nos outils le sous ensemble des machines/logiciels à mettre à jour.

3. Minerve : vérification de l'applicabilité dynamique des correctifs

Les rustines sources fournies par les éditeurs logiciels sont générées et déployées à partir des programmes `diff` et `patch`. Ces outils manipulent uniquement des différences textuelles entre fichiers sources sans exploiter des informations sémantiques

sur les modifications apportées. Si cette approche permet de valider l'application d'un correctif dans un cadre statique (ou off line), elle n'est pas suffisante dans un cadre dynamique (ou on line). Minerve est chargé de classifier les changements effectués dans le code de l'applicatif puis d'informer le RSSI de l'applicabilité dynamique de ceux-ci (section 3.1). Enfin, un fichier des modifications plus expressif qu'un simple `diff` est produit pour faire appel à l'audit du RSSI (section 3.2).

En vue de démontrer la faisabilité de notre approche nous nous sommes limité à ne considérer que les programmes écrits en langage C ANSI et compilé sur les plateformes IA-32 dans un environnement Linux. Nous supposons également que les rustines sources fournies par les éditeurs sont cohérentes (i.e. que les sources corrigées passent la phase de compilation et fournissent un programme valide).

3.1. Classification et Vérification

Le premier objectif de l'outil est de classifier les modifications à effectuer sur un programme cible écrit en C. Le langage C est un langage procédural typé à effet de bord. Il contient deux sortes de types, les types simples qui reflètent les entités que le processeur peut manipuler efficacement, et les types structurés, construits par le développeur par agrégation de types simples. Une rustine source modifie le comportement d'un programme en modifiant la définition des procédures qu'il contient, et/ou en modifiant les types associés aux variables qu'il utilise. Si d'un point de vue statique les deux types de modifications peuvent être traitées symétriquement, d'un point de vue dynamique, chacun des cas doit être traité différemment.

3.1.1. Modification de fonctions

Nous distinguons trois types de correctifs liés aux fonctions : l'ajout, la suppression et la modification. En réalité ces trois types de correctifs peuvent se résumer en un seul : la modification d'une fonction. En effet, l'ajout d'une fonction g dans un programme implique qu'il existe au moins une fonction f qui a été modifiée de sorte qu'elle utilise g . Dans le cas contraire l'ajout de g n'aurait aucun sens. De même, la suppression d'une fonction g nécessite qu'au moins une fonction f soit modifiée. Dans le cas contraire, une erreur à la compilation du programme corrigé serait générée. Or, nous avons précisé que les rustines considérées étaient cohérentes. Nous nous limitons donc aux cas de la modification d'une fonction f et parlerons de f' pour désigner la nouvelle version de f (donc après application de la rustine).

La modification d'une fonction peut entraîner le changement de sa signature et/ou le changement de son corps. Le changement de la signature d'une fonction est équivalent à l'ajout d'une nouvelle fonction. Pour le deuxième cas, le remplacement de f par f' sans modification de prototype, il est nécessaire de s'assurer que f n'est et ne sera jamais plus exécutée [SEG 93, GUP 98]. Cette première condition ne peut être vérifiée que dynamiquement par inspection de la pile d'exécution. Pour cela, nous nous appuyons sur l'outil Arachne présenté en section 4. De plus, le code ajouté (si il

y a lieu) dans f ne doit pas lire les données produites par f . Pour cela nous pensons utiliser un outil de propagation de type nommé Lackwit [O'C 97].

3.1.2. *Modification d'une variable*

Nous distinguons deux types de modifications, celles liées au changement de type de base, et celles liées à la modification des types structurés.

- La redéfinition du type d'une variable nécessite deux opérations : la propagation de l'état de la variable et les modifications à apporter au code qui la manipule. Augmenter la taille d'une variable sans modifier son type numérique (ex : $int \rightarrow long\ int$) ne pose pas de problème de conversion, puisque le nouveau type englobe l'ancien. Au contraire, la diminution de la taille d'une variable (ex : $long\ int \rightarrow int$) ou la modification du type numérique de celle-ci (ex : $float \rightarrow int$) n'est possible que si sa valeur courante peut-être contenue par le nouveau type ou si l'on dispose d'une fonction de conversion. Modifier la taille ou le type numérique d'une variable nécessite de modifier également le code binaire manipulant celle-ci. En effet, les instructions processeur, les registres, les drapeaux ainsi que les exceptions associées à des types différents sont distincts. La modification d'une instruction peut nécessiter de réorganiser les instructions voisines. Cette réorganisation peut s'étendre au remplacement complet de la fonction accédant à la variable dont le type a été modifié. Les registres globaux étant de la responsabilité des fonctions appelantes et les autres registres des fonctions appelées, la modification du type numérique d'une variable peut alors également nécessiter la modification des fonctions appelant les fonctions qui accèdent aux variables modifiées. Ce cas n'est néanmoins pas un problème puisqu'il est résolu par la technique d'instrumentation d'Arachne.

- Modification de la définition d'un type structuré : nous avons choisi de présenter l'ajout d'un champ dans un type structuré puisqu'il illustre bien les problèmes liés à ce type de modifications (suppression, remplacement, inversion). La modification d'un type structuré peut modifier la contrainte d'alignement sur la représentation mémoire des instances de ce type. Certaines instructions de la plate-forme IA-32 peuvent avoir des comportements différents, voir ne pas fonctionner lorsque les données qu'elles manipulent ne sont pas alignées. Modifier toutes les variables existantes du type modifié imposerait alors également de mettre à jour l'ensemble des instructions manipulant ce type dans le programme d'origine. Or, la cohérence de la mise à jour ne pourrait alors être garantie qu'en stoppant le programme. Toutefois, une technique de réécriture appropriée permet de résoudre ce problème. En effet, un champ ajouté dans un type structuré lors d'une mise à jour n'existe pas dans le programme de base, seules les instructions ajoutées peuvent manipuler celui-ci. Il est ainsi possible de modifier ces dernières de façon à associer via une fonction de hachage, à chaque variable du type d'origine une variable représentant le nouveau champ. La création du nouveau champ peut se faire de manière paresseuse pour chaque variable. Ceci nous permet d'éviter la modification de toutes les variables de ce type lors de la mise à jour.

Sachant que le RSSI est placé au centre de notre architecture, l'ensemble des informations de vérifications effectuées par Minerve ne lui sont présentées qu'à titre indicatif.

Libre à lui de valider ou non la rustine dynamique. Pour l'aider dans cette tâche Minerve produit un fichier concentrant l'ensemble des modifications à apporter au logiciel incriminé dans un format lisible et compatible avec notre outil de transformation de code à chaud. La section suivante décrit le processus d'inspection de la rustine.

3.2. *L'audit du RSSI*

L'inspection du code de la rustine répond à deux buts. D'abord elle permet de s'assurer que la faille de sécurité recensée est véritablement corrigée. Puis, elle veille à ce que de nouveaux trous de sécurité ne soit pas introduits par erreur. Il peut également être utile d'adapter un correctif de sécurité au système local d'information. Par exemple, différents experts en sécurité préconisent que la correction d'une vulnérabilité s'accompagne de la mise en place d'une alarme signalant toute tentative d'exploitation de la vulnérabilité colmatée [SCH 00]. Une fois la rustine traduite par Minerve dans le langage d'Arachne, le RSSI peut facilement l'éditer au cours de l'audit afin d'y placer une alarme qui va par exemple, déclencher le système de détection d'intrusion (IDS).

Au contraire de `diff` et `patch` qui fonctionnent par substitution de lignes, Minerve utilise Arachne pour remplacer des fonctions aux comportements erronés par de nouvelles fonctions et redéfinir les accès fautifs aux variables globales. Une fois la rustine traduite par Minerve dans le langage d'Arachne, la rustine contient l'ensemble des entités (variables globales et fonctions) à redéfinir. Chaque rustine de sécurité est ainsi traitée comme une préoccupation à part, inspectable et éditable isolément des autres. De plus, le format de fichier est directement compatible avec Arachne [SEG 03a], un tisseur dynamique d'aspect pour le langage C.

Avec Arachne, chaque substitution ou *aspect* est définie par un couple (coupe, action). Une coupe spécifie quand dans le flot d'exécution du programme, le code décrit dans l'action doit s'exécuter. Une coupe est une séquence d'appels de fonctions éventuellement terminée par un accès en lecture ou en écriture à une variable globale. Tous les éléments, sauf le dernier de la séquence, déterminent le contexte d'exécution de la coupe. Dans ce cadre d'étude, nous utilisons Arachne comme un outil d'instrumentation à chaud nous permettant d'intercepter tout accès à une fonction ou à une variable particulière. De ce fait, les abstractions langage de haut niveau propre à la programmation par aspect et fournies par Arachne ne seront pas utilisées dans notre solution.

4. Arachne : injection dynamique des correctifs

Cette section décrit les outils associés à Arachne permettant au RSSI de compiler la rustine dynamique avant de l'injecter dans l'application à mettre à jour, puis s'achève par une vue d'ensemble des mécanismes exploités dans la mise en œuvre d'Arachne.

4.1. Les outils de compilation et de déploiement

Arachne s'accompagne de deux outils, un compilateur d'aspects `acc`, générant une DLL native et un tisseur `weave` permettant d'injecter, *tisser* les aspects compilés dans l'application incriminée. Indépendamment de notre hypothèse de travail qui suppose que la rustine traduite par Minerve est correcte, la compilation effectuée par `acc` constitue une vérification globale de la cohérence de la rustine. Plus précisément, elle garantit l'absence d'erreurs syntaxiques. De plus, à l'injection, `weave` vérifie que l'application à mettre à jour contient bien les symboles attendus par la rustine. Ainsi, si une rustine essaie de remplacer une fonction ou une variable globale qui n'est pas définie dans l'application, l'injection de la rustine échoue. Bien que ce type de vérification ne permette pas de garantir que la rustine corrige le trou de sécurité, il permet néanmoins d'assurer un niveau minimal de cohérence. Si la rustine nécessite plusieurs points d'instrumentation, Arachne garantit l'atomicité de la réécriture : soit toutes les fonctions/variables sont remplacées soit aucune. Enfin, une option particulière permet à Arachne de vérifier si une fonction à remplacer est en cours d'exécution et dans l'affirmative, d'attendre la fin d'exécution avant le tissage.

4.2. Implémentation d'Arachne

Le tisseur d'Arachne, piloté par la commande `weave`, réécrit à la volée le code binaire de l'application dans laquelle il doit injecter les aspects [SEG 03b]. Il est difficile de couvrir les différentes stratégies de réécriture utilisées par le tisseur. Nous précisons ici toutefois le principe général qui permet à Arachne de remplacer une fonction par une autre avant de préciser les problèmes techniques qui compliquent sa mise en oeuvre.

Sur une machine Pentium sous Linux, un appel de fonction au niveau source se traduit dans le code binaire de l'application par une instruction assembleur `CALL` suivie de l'adresse de la fonction à invoquer. Arachne désassemble le code binaire de l'application et y recherche les instructions `CALL`. Pour cela, il utilise les tables de symboles générées à la compilation pour retrouver, à partir de l'adresse manipulée par l'instruction `CALL`, le nom symbolique de la fonction invoquée. Au tissage, Arachne charge la DLL contenant les aspects et réécrit les différents `CALL` vers la fonction à remplacer par des `CALL` vers la fonction appropriée de la DLL. Un mécanisme similaire mais plus complexe permet à Arachne de remplacer les lectures/écritures sur les variables globales.

En pratique, plusieurs problèmes compliquent la mise en oeuvre du principe de réécriture décrit plus haut. En résumé, il s'agit de garantir l'atomicité des réécritures effectuées dans le code objet de l'application, d'assurer la cohérence de l'exécution, de garantir que l'ensemble des aspects de la DLL soit tissés d'une manière atomique (c'est-à-dire qu'un aspect ne puisse pas s'exécuter alors que les autres sont en train d'être tissés), de gérer les contraintes imposées par le système d'exploitation en terme

de séparation des espaces d'adressage et finalement de résoudre les problèmes d'efficacité. Arachne résout les problèmes d'atomicité des réécritures par un système de verrous tournants reposant sur la taille des instructions de sauts courts et sur la taille des accès mémoire atomiques sous Pentium. La cohérence de l'exécution est assurée par un crochet généré à la volée lors du tissage : Ce crochet sauvegarde et restaure les registres nécessaires. L'utilisation d'un test sur une garde garantit que l'ensemble des aspects de la DLL sont tissés de manière atomique. La séparation imposée des espaces d'adressage par le système d'exploitation empêche un processus de modifier l'espace mémoire d'un processus tiers. La première fois qu'Arachne tisse dans un processus, il accède à son espace mémoire en utilisant les fonctions de débogage `ptrace`, puis il crée un processus léger d'instrumentation partageant le même espace d'adressage que l'application. Arachne pilote ensuite ce processus léger d'instrumentation en utilisant une interface d'interconnexion : le tissage s'effectue ainsi globalement sans interruption de l'application. Arachne adresse les problèmes d'efficacité par des caches conçus pour n'examiner une fois au plus le code binaire de l'application.

5. Évaluation

Cette section présente une évaluation de notre atelier composé de Minerve et Arachne. Pour cela, nous avons appliqué notre méthodologie aux mises à jour de sécurité concernant des logiciels libres, publiées depuis le début de l'année 2002 par le CERT/CC. Après une présentation de l'organisme et de l'échantillon d'étude, nous présentons nos résultats sur l'ensemble de l'échantillon puis décrivons un exemple concret de rustine dynamique.

5.1. Échantillon d'étude

Le CERT/CC pour « Computer Emergency Response Team / Coordination Center » est un centre d'expertise créé en novembre 1988, suite à l'apparition du ver *Morris*. Par ses activités de veille, de diffusion, d'alerte et de formation, il cherche à améliorer la sécurité des systèmes informatiques. Sa longévité, son indépendance vis-à-vis des éditeurs de logiciels ainsi que son rôle préventif et curatif font de lui un acteur de référence mondiale du domaine. Depuis 1988, il a ainsi collecté une des bases de données les plus complètes en matière de failles de sécurité. Toutes les failles relevées par le CERT/CC sont classées suivant trois catégories. La première, les alertes recensent les attaques perpétrées contre des systèmes informatiques. La seconde, les rapports de vulnérabilité listent les failles dans les applications. Enfin la dernière catégorie, les avis référencent un ou plusieurs rapports de vulnérabilité. Les avis se limitent aux failles de sécurité critiques sur des logiciels largement répandus et fournissent systématiquement les correctifs associés.

Notre étude s'est donc portée sur l'ensemble des avis publiés par le CERT/CC depuis le début de l'année 2002 jusqu'à aujourd'hui. Cette période représente un total de 67 avis, 30% concernent des produits Microsoft, 20% des logiciels propriétaires

autres que Microsoft, 40% référencent des failles dans des logiciels libres et les 10% restants concernent de l'informatique embarquée dans des matériels réseau. 14 avis concernent des logiciels libres sous Linux. Nous en avons écarté deux parce qu'ils affectaient des versions obsolètes introuvables.

Nous avons constaté que 44% des failles examinées sont des débordements de tampon : Ce type de dysfonctionnement peut permettre à un utilisateur malveillant de faire exécuter du code arbitraire par le processus visé. 19% présentaient des erreurs de format : ces erreurs affectent les fonctions à nombre de paramètres variables (ex : `printf`). Un utilisateur malveillant peut exploiter cette faille pour obtenir des informations sur le processus visé. 12% des cas recensent des doubles désallocations mémoire. Cette faille provoque normalement une erreur de segmentation mais peut sous certaines conditions permettre l'exécution de code arbitraire. 6% des avis examinés présentent un débordement d'entier : Ceci se produit lorsque la valeur d'un entier dépasse la limite du type. Enfin, 19% des trous de sécurité résultent de combinaisons des erreurs de conceptions citées ci-dessus.

Toutes ces failles se basent sur des hypothèses faites par le programmeur sur les paramètres du programme mis en défaut par un utilisateur malveillant. Ainsi, ces failles peuvent se corriger très simplement : il suffit d'ajouter des tests sur les valeurs des paramètres. L'examen des rustines correctives de ces trous de sécurité nous a permis de dégager le contenu des modifications apportées à un programme pour corriger une faille. Nous avons pu constater la corrélation avec le type des dysfonctionnements recensés, puisque 90% des correctifs source ne contenaient que des modifications de fonctions sans changement de signatures. Seulement 10% contenait à la fois des modifications de code et des transformations de types.

Ce constat ainsi que les expérimentations que nous avons réalisées, nous permettent d'affirmer que notre atelier est applicable à tous les trous de sécurité étudiés.

5.2. Un exemple de mise à jour

Nous présentons dans cette section une des failles que nous avons pu corriger automatiquement grâce à Minerve. Il s'agit d'une vulnérabilité découverte en juin 2002 et qui concerne le serveur de communication OPENSSH (avis numéro 18/2002 du CERT/CC) [CER 03]. Cette faille concerne le mécanisme d'authentification du protocole SSH2 : `challenge response`. Elle affecte les versions de 2.3.1p1 à 3.3 configurées avec les options `SKEY` ou `BSD_AUTH`. C'est le cas pour l'installation par défaut dans la plupart des distributions Linux. Cette faille consiste en un débordement d'entier exploitable par des utilisateurs malveillants afin d'exécuter du code arbitraire par le programme résidant `sshd`, exécuté généralement avec les droits super-utilisateur.

5.2.1. Rustine source

La rustine source fournie par les développeurs de OPENSSH corrige les fonctions `input_userauth_info_response` et `input_userauth_info_response_`

pam. Les modifications se bornent à ajouter des tests de validité sur le paramètre `nresp` dans ces deux fonctions. Lorsque ces tests de validité échouent, la fonction `fatal` est déclenchée pour terminer la session en cours. Comme le montre la figure 1, l'examen de la rustine produite par `diff` ne permet pas de comprendre facilement les changements effectués par le correctif.

```

    authctxt->postponed = 0; /* reset */
    nresp = packet_get_int();
+  if (nresp > 100)
+    fatal("input_userauth_info_response : nresp too big %u", nresp);
    if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (i = 0; i < nresp; i++)

```

Figure 1 – Une partie du fichier de différences `diff` de la mise à jour de sécurité de `sshd` référencée par le CERT/CC par CA-2002-18

5.2.2. Rustine dynamique

Minerve produit à partir de la rustine source une rustine dynamique. La rustine dynamique contient essentiellement une suite d'aspects qui réalisent le remplacement des fonctions du programme de base, `input_userauth_info_response` et `input_userauth_info_response_pam`. Les fonctions corrigées possèdent le suffixe `"_new"`. Finalement, les anciennes fonctions pouvant être appelées dans `sshd` indifféremment par un appel direct de fonction ou par un appel par pointeur, deux aspects sont nécessaires pour chacune de ces fonctions afin de réaliser leur remplacement à coup sûr. La figure 2 illustre ces aspects pour la fonction `input_userauth_info_response`.

La coupe de l'aspect `ReplaceFunctionCall` capture l'ensemble des appels directs à cette fonction (ligne 2 du source figure 2). L'action remplace son exécution par celle de la nouvelle version à laquelle les mêmes arguments sont fournis (ligne 3). Cet aspect permet ainsi de remplacer tous les appels directs à `input_userauth_info_response`. La coupe du second aspect `ReplacePointer` capture les accès en lecture sur l'adresse de la fonction `input_userauth_info_response` (ligne 6). C'est cette valeur qui peut être ensuite utilisée dans un appel de fonction par pointeur. La valeur lue est remplacée dans l'action de l'aspect (ligne 7) par l'adresse de la nouvelle version de la fonction définie dans la rustine dynamique. Ainsi toutes les lectures de l'adresse de l'ancienne fonction seront remplacées par une lecture de l'adresse de la nouvelle fonction.

Comme le montre la figure 2, la syntaxe des rustines produites par Minerve facilite réellement le travail d'audit des correctifs. Sur l'ensemble des correctifs publiés par le CERT/CC en 2002, le gain de temps est réellement sensible. Si on exclut les temps réseau, dans nos expériences, une mise à jour est déployée en moyenne sur un poste

```

1 : ReplaceFunctionCall [ :
2 :   void input_userauth_info_response(int type, u_int32_t seq, void* ctxt) [ : {
3 :     input_userauth_info_response_new(type, seq, ctxt); } :]
4 :
5 : ReplacePointer [ :
6 :   void* (input_userauth_info_response)(int, u_int32_t, void*) [ : {
7 :     return &input_userauth_info_response_new; } :]

```

Figure 2 – Aspects générés par Minerve pour le remplacement systématique d’une fonction

en 253 μs sans même que l’utilisateur en ait conscience. De plus avec notre atelier, le RSSI ne gère pas la mise à jour d’un poste mais de la totalité du parc. Le temps pour appliquer un correctif est indépendant du nombre de machines composant le parc.

6. Travaux apparentés

À notre connaissance, aucune autre solution ne propose *à la fois* la vérification de l’applicabilité d’une mise à jour dynamique et son déploiement à chaud. La vérification et la cohérence des mises à jour ont essentiellement été étudiées sous l’angle de contraintes sur l’état de l’exécution de l’application à mettre à jour [GUP 96] ou sur la structure [SEG 93] de son code. Notre approche se concentre elle sur la structure de la rustine source traditionnellement distribuée. Dyninst [BUC 00] et Vulcan [EDW 01] explorent tout deux la possibilité de réécrire le code binaire à chaud. Contrairement à notre approche, ces outils travaillent au niveau de l’assembleur et ne garantissent pas l’atomicité de la réécriture. C’est un problème quand la mise à jour implique d’instrumenter le programme à plusieurs endroits.

7. Conclusion

Nous avons présenté dans cet article un atelier pour le déploiement à chaud de correctifs de sécurité. Cet atelier est composé de deux outils. Le premier, Minerve, assure la faisabilité d’un déploiement à chaud et réduit le temps nécessaire à la certification par le RSSI d’un correctif. Le second, Arachne, applique à chaud sur un programme en cours d’exécution, la rustine dynamique produite par Minerve. Notre atelier s’insère dans le processus traditionnel de mise à jour des logiciels libres basé sur les utilitaires `diff` et `patch`. L’utilisation de notre atelier permet un gain de temps significatif dans le déploiement et apporte donc une plus grande réactivité quant à la correction des failles.

Concrètement, Minerve, transforme un correctif peu lisible en un fichier décrivant les modifications à apporter au logiciel. Ces modifications sont exprimées sous forme d’un aspect dont les coupes se limitent au remplacement de fonctions et/ou à l’accès à

une variable. En ce sens nous n'utilisons pas l'ensemble du pouvoir d'expression de la programmation par aspect. Le second, Arachne, injecte dynamiquement l'aspect compilé dans l'application à mettre à jour. Pour des raisons techniques, nous nous sommes pour l'instant limités aux programmes écrits en C s'exécutant sur une plate-forme IA32 dans un environnement Linux. L'atelier Minerve/Arachne a permis de déployer l'ensemble des correctifs de sécurité publiés par le CERT/CC sur les logiciels libres depuis 2002.

8. Bibliographie

- [BUC 00] BUCK B., HOLLINGSWORTH J. K., « An API for Runtime Code Patching », *The International Journal of High Performance Computing Applications*, vol. 14, n° 4, 2000, p. 317–329.
- [CER 03] CERT, « Computer Emergency Response Team Coordination Center », rapport, 2003, Carnegie Mellon University.
- [DEV 04] DEVOTEAM, « Enquête Européenne sur la Sécurité des Systèmes d'Information », rapport, 2004, XP Conseil.
- [EDW 01] EDWARDS A., SRIVASTAVA A., VO H., « Vulcan », rapport n° MSR-TR-99-76, janvier 2001, Microsoft Research (MSR).
- [GUP 96] GUPTA D., JALOTE P., BARUA G., « A Formal Framework for On-line Software Version Change », *IEEE Transactions on Software Engineering*, vol. 22, n° 2, 1996, p. 120–131.
- [GUP 98] GUPTA D., BARUA G., JALOTE P., « A Formal Framework for On-line Software Version Change », , 1998.
- [HP 04] HP, « NOVADIGM : Software management, automated change management », rapport, 2004, Hewlett-Packard.
- [O'C 97] O'CALLAHAN R., JACKSON D., « Lackwit : A Program Understanding Tool Based on Type Inference », *Proceedings of the 1997 International Conference on Software Engineering*, ACM Press, 1997, p. 338–348.
- [SCH 00] SCHNEIER B., *Secrets and Lies : Digital Security in a Networked World*, Wiley, août 2000.
- [SEG 93] SEGAL M. E., FRIEDER O., « On-the-Fly Program Modification : Systems for Dynamic Updating », *IEEE Software*, vol. 10, n° 2, 1993.
- [SEG 03a] SEGURA-DEVILLECHAISE M., MENAUD J.-M., LAWALL J., MULLER G., « Extensibilité Dynamique dans les Caches Web : une Approche par Aspects », *3ème Conférence Française sur les Systèmes d'Exploitation (CFSE'03)*, La Colle sur Loup, octobre 2003.
- [SEG 03b] SEGURA-DEVILLECHAISE M., MENAUD J.-M., MULLER G., LAWALL J., « Web Cache Prefetching as an aspect : Towards a Dynamic-Weaving Based Solution », *Proceedings of the 2nd international conference on Aspect-oriented software development*, Boston, Massachusetts, USA, mars 2003, ACM Press, p. 110–119.